DOCUMENT RESUME

ED 419 502 IR 018 913

AUTHOR Kurtz, Barry; O'Neal, Michael
TITLE Developing Educational Materials in Java: A Report from the Front Lines.
PUB DATE 1998-00-00
NOTE 15p.; In: NECC '98: Proceedings of the National Educating Computing Conference (19th, San Diego, CA, June 22-24, 1998); see IR 018 902.
PUB TYPE Reports - Descriptive (141) -- Speeches/Meeting Papers (150)
EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS College Instruction; *Computer Assisted Instruction; Computer Science; *Computer Simulation; Computer Software Development; Higher Education; *Instructional Materials; Interaction; *Internet; Material Development; Problem Solving
IDENTIFIERS *Java Programming Language

ABSTRACT

This paper describes the use of Java to develop a variety of educational materials to supplement both traditional instruction and Internet-based instruction. Efforts have focused on three projects that vary in course level, content, and style of interaction. Unlike the simple Java applets on the Web, these are very sophisticated simulation environments that are at the cutting edge of Java development. The freshman Overview of Computer Science course uses a set of elaborate Java simulations in a closed lab setting where students work in small groups to explore a problem domain while the instructor circulates among them providing assistance as necessary. The Concurrency Simulator used in the sophomore level Introduction to Parallel Programming course is used for demonstration during classroom instruction and for students to complete programming assignments outside of class. The Operating Systems course uses Java applets to provide student interaction in an Internet-based course. After presenting an overview for each of these projects, the Java programming environment is discussed, including successes and problems encountered during implementation, common factors in the design of materials, measuring the educational effectiveness of the materials, and advice on the development of Java-based educational materials. (Author)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED 419 502

Paper Session

Developing Educational Materials in Java: A Report From the Front Lines

Barry Kurtz
Louisiana Tech University
Ruston, LA 71272
318.257.2436
kurtz@coes.latech.edu

Michael O'Neal
mike@coes.latech.edu

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
☐ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
D. Ingham
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

Key Words: Java, simulation, computer-based education

Abstract

We have been using Java for the past two years to develop a variety of educational materials to supplement both traditional instruction and Internet-based instruction. Our efforts have focused on three projects that vary in course level, content, and style of interaction. Unlike the simple Java applets you see on the Web, these are very sophisticated simulation environments that are at the cutting edge of Java development.

Our freshman Overview of Computer Science course (hereafter referred to as CS100) uses a set of elaborate Java simulations in a closed lab setting where students work in small groups to explore a problem domain while the instructor circulates among the students providing assistance as necessary. The Concurrency Simulator used in our sophomore level Introduction to Parallel Programming course (hereafter referred to as CS240) is used for demonstration during classroom instruction and for students to complete programming assignments outside of class. Our Operating Systems course (hereafter referred to as CS345) uses Java applets to provide student interaction in an Internet-based course.

After providing an overview for each of these projects we discuss the Java programming environment, successes and problems encountered during implementation, common factors in the design of materials, measuring the educational effectiveness of the materials, and advice to others on the development of Java-based educational materials.

The Watson Project

In 1992 the authors were awarded a grant by the National Science Foundation (DUE 9254317) to develop a collection of "hands-on" laboratory experiences to support a

IR018913

breadth-first introduction to computing. The project was based on the recognition that while many schools were beginning to offer breadth-first introductions to computing in response to the Denning report (Denning et al., 1989), there was (and continues to be) little in the way of software available to support these courses.

Topics covered in Louisiana Tech's CS100 class can be grouped into four broad categories: end-user applications, software development, architecture and digital logic, and the limitations and potential of computing. The authors believed that if freshmen-level students were to gain any real insight into such a vast array of topics most would need some form of hands-on experience. Hence, the authors launched the Watson project—so named because its modules are supposed to assist the student in learning fundamental computing concepts. Given the real life experiences of Alexander Graham Bell and the fictional exploits of Sherlock Holmes, it seemed that "Watson" would be the perfect name for our "assistant."

In its original incarnation, Watson consisted of nine independent modules written in C using SUIT (Simple User Interface Toolkit) (Conway, 1992). These modules included:

- a spreadsheet lab
- a relational database lab
- a data structures lab focusing on stacks and queues
- a specialized imperative programming language for drawing graphical objects
- a more general Pascal-like imperative language
- a functional language, based on Lisp
- an assembly language and machine organization lab
- a digital logic lab
- a finite state automata lab

The original Watson labs were used at Louisiana Tech University for over three years (Kurtz, 1994; O'Neal, 1995). While the labs were very popular with the students, they were never robust enough, despite our best efforts, to be exported to other schools. The primary reason for this was that Watson had been constructed on top of SUIT, a pre-Java "platform independent" interface library. Since one of the project goals was portability of the software, SUIT seemed like an excellent choice—it promised to allow Watson to run on PCs, Macs, and Unix workstations. As the project progressed, however, it became clear that SUIT was not up to the task. In fact, we could never get SUIT to behave reliably on the most popular platform of the day, Windows 3.1, even though we devoted substantial resources to completing a port of SUIT to Windows 3.1

With the release of Java, it became obvious that if Watson was ever to have an impact beyond Louisiana Tech the labs would have to be re-implemented in Java. Thus, beginning in the summer of 1996, we began a major push to rewrite Watson from scratch in Java. At the present time we have classroom tested the Java versions in six of the nine labs. A seventh lab, the imperative programming lab, is under active development. An eighth lab, based on Prolog, is in the initial design stage. The functional and finite automata labs have not been ported to Java.

A great deal of effort went into designing a "look and feel" for Watson that is consistent throughout the labs. The process of deciding on a set of interface

guidelines is never easy, but was complicated in our case by the large variety of laboratory experiences we wanted to present to the students, ranging from spreadsheets and databases to digital circuit design. Our goal throughout this process has been to create an environment that is both intuitive and easy to use.

We have observed that many beginning students feel uncomfortable in front of a keyboard. The mechanics of typing and entering commands and correcting mistakes are difficult for some. A more widely recognized problem is the difficulty students have with programming language syntax. Our approach to these problems is to use syntax directed editors for programming languages, so that only syntactically correct programs can be entered, and to limit keyboard input wherever practical. For example, most of our laboratory experiences require no keyboard input at all, only mouse presses. A few of the labs require limited keyboard input, such as the imperative programming lab, where the name of an identifier must be entered when it is first declared.

In order for Watson to gain the widest possible acceptance, it was deemed important that the Java-based labs be accessible through the Web. All of the modules discussed below were developed under Version 1.02 of Java and thus run under Netscape's Communicator 4.0 and Microsoft's Internet Explorer 3.0. These labs are available at http://www.LaTech.edu/~watson/. The one exception to this rule is the imperative lab, which is being written in Version 1.1 of Java—a version that is supported only by the latest versions of Web browsers.

Spreadsheet. The spreadsheet laboratory allows students to enter numbers, formulas, and text into a simple spreadsheet. The spreadsheet supports basic arithmetic operations, such as addition, subtraction, multiplication, and division; and built-in functions, such as summation and average. This is the first lab completed by the students and, as such, introduces them to the look and feel of all the Watson modules.

Database. Students study relational concepts using an academic database that contains tables of student information, faculty information, and course-scheduling information. The fundamental operations of project, select, and join are first introduced in a Query by Example (QBE) mode. Students are asked to use QBE to solve particular queries, such as to list the name and gender of all students who earned a C or higher in computer science courses. At the same time QBE is being performed, the corresponding relational equations appear on the screen. After becoming familiar with relations and queries using QBE, students are asked to perform queries using relational equations directly.

Data Structures. Students study algorithms to manipulate linear structures, such as stacks and queues, and tree structures. These data objects are shown on the screen and manipulated directly using buttons labeled "push," "pop," "enqueue," "dequeue," and so forth. A typical problem is to reverse a queue, which requires students to push all of the items in the queue onto a stack and then pop the items off the stack and put them back in the queue.

Graphics. The graphics laboratory has three major divisions of the screen: a drawing window, an object declaration window, and a program code window. Objects supported by the lab include points, lines, circles, polygons, and the abstract concept

of distance. Commands include assign, draw, erase, color, loop, increment, and decrement. Initially students draw objects directly in the drawing window using mouse operations similar to those supported by most drawing packages. As objects are drawn in the drawing window, the corresponding declarations and program code automatically appear in the appropriate windows. In later exercises, students must first enter the declarations and commands (using a syntax-directed editor) that will be executed to draw a picture. By the time students complete this lab, they should have an understanding of data types, variables, constants, assignment, output, command sequencing, and simple repetition. A screen snapshot from the graphics lab is shown in Figure 1.

Figure 1. The Watson Graphics Laboratory

Imperative Programming. Imperative programming is introduced using a Pascal-like language in a tightly constrained lab environment where only syntactically correct programs, including type checking, can be entered. In addition to repetition and sequencing covered in the graphics lab, the imperative laboratory introduces the concepts of selection, using an if command, and procedure encapsulation, with parameter passing to transfer information. There are only two data types in the language: integer and string.

Assembly Language and Machine Organization. This lab involves a simple register-based computer with a 16-instruction machine language. Students construct assembly language programs using a mouse-oriented, syntax-directed editor. The lab shows the state of the CPU, including register contents, program counter, and flag bits. In addition, the contents of memory are visible. To help students become familiar with various number bases, the lab has two modes: binary and hexadecimal.

Digital Logic. Three basic gates are available in this lab, a two-input AND, a two-input OR, and a single-input NOT. Students construct circuits by manipulating on-screen gate symbols that can be connected together using the mouse. The types of problems solved include construction of XOR, adders, data selectors, encoders, and decoders. The lab automatically generates truth tables and Boolean equations from the students' circuits.

The Concurrency Simulator

BEST COPY AVAILABLE

The Concurrency Simulator is used in a sophomore-level course, CS240, to introduce parallel computing early in the undergraduate curriculum (Kurtz et al., 1998a). The course and software was developed as part of a National Science Foundation grant (CDA-9414309) awarded to the authors in 1994 (Kurtz et al., 1998b). This simulator is unique since it integrates a graphical topology into the programming environment. This makes algorithms much easier to express and, combined with a context-sensitive editor, makes it easy for students to enter their algorithms without becoming bogged down in the details of language syntax.

There were two primary inspirations for the development of the Concurrency Simulator. One of the textbooks we used in the course (Andrews & Olsson, 1993) gave three alternative solutions to the dining philosophers problem; each solution was presented using an accompanying diagram, as shown in Figure 2, to explain the code. The centralized solution to the left has one fork server surrounded by five philosophers. The decentralized solution in the middle has a server for each fork in between each pair of philosophers. The decentralized solution to the right has a server for each philosopher; these servers communicate with each other about the forks. We designed the Concurrency Simulator to use a topology similar to these diagrams as a starting point for the development of an algorithm in an integrated programming environment.

The second inspiration came from Parallaxis, a programming environment for SIMD simulations (BraŸnl, 1993). After developing a Parallaxis program you have the option of displaying the program execution graphically using an appropriate topology (e.g., grid or ring) with variable values shown in a range of colors. We combined these approaches in our Concurrency Simulator. The topology of the problem solution is an integral part of the programming environment, and it initiates the development of an algorithmic solution. At run time this topology illustrates program execution by changing colors of the program components based on their current execution state.

Figure 2. Topologies for the Dining Philosophers

Our Concurrency Simulator can be used to illustrate three approaches to parallel algorithm development: semaphores, monitors, and rendezvous (Ben-Ari, 1982). All three approaches share a common look and feel as shown in Figure 3.

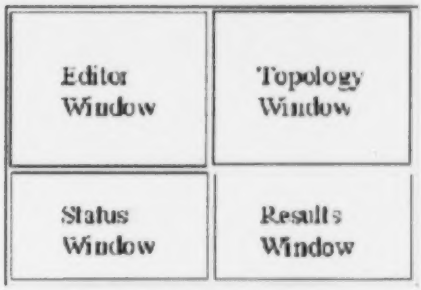


Figure 3. Screen Layout

There are four windows on the screen: the top two are used for program entry and the bottom two display program execution. The student first specifies the topology. The objects vary depending on the paradigm:

- Processes and semaphores
- Processes and message links for rendezvous
- Processes and monitors with entries and condition variables

Clicking the right mouse button on an object displays a popup menu with options to copy an object, show code for the object, or delete an object. Objects are named by a single letter and a digit; the digit is automatically assigned as each new instance of that object is created. For example, the process associated with a philosopher may be named A and numbered A0, A1, A2, and so forth.

The editor window displays:

- Source code for the processes
- Source code for the monitor entries
- Declarations and initialization for the message links, semaphores, or condition variables, as appropriate

The program is entered in a point-and-click environment where the only keyboard entry is for literal values or the names of objects when they are declared. The student has the choice of mouse entry using a virtual keyboard or typing directly on a physical keyboard. This editor is context sensitive. Only proper types are allowed, and when variables are selected it is from a popup menu of all variables of the appropriate type that are currently visible. A process is defined starting with a basic skeleton with placeholders for <declarations> and for <statements>. Clicking on a placeholder brings up a menu of acceptable substitutions, which are either combinations of source code and other placeholders or simply source code. Program development continues until there are no placeholders left to be specified. All program code is syntactically correct after entry is completed. The topology can be modified or expanded as the algorithm is fully developed

Output as a result of Write statements appears in the results window as the program executes. The status window shows the various states of processes in written form. For example, for monitors the queue of waiting processes for all entry points and condition variables are shown. At the same time the topology displays the status of

processes through color changes. For example, when a process is executing outside the monitor, it appears yellow. When a process is inside the monitor and executing, both the process and the entry point are green. When a process is blocked on an entry point or a condition variable, both objects are displayed in red. A sample screen for the monitor lab is shown in Figure 4.

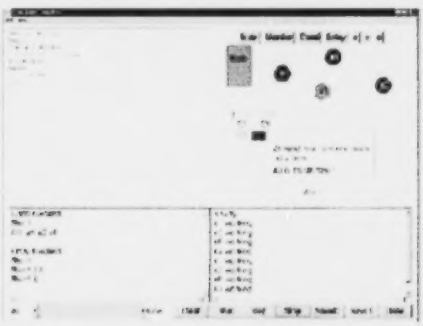


Figure 4. Screen image for monitor lab

Although the implementation details for the Concurrency Simulator are beyond the scope of this paper, we can give a general idea of the complexity by briefly describing two components. The context-sensitive editor allows the user to specify in data files the language syntax as well as the options on the popup menus of choices. The editor reads in these components and creates an appropriate user interface. An attribute-like mechanism is used to create the context sensitivity (Reps, 1989; Slonneger, 1995). Using this general mechanism made it easier to create distinct editors for each of the paradigms without having to redo the internals of the editor code.

The runtime environment for the simulator is threaded with a language interpreter running for each process shown on the topology. Consider the Dining Philosophers topology shown in the center of Figure 2. There are 10 interpreters running simultaneously, one for each of the philosophers and one for each of the forks. There is a generalized interconnection network underlying these processes that handles the message passing via the links. Despite this complexity, performance has been acceptable on all Pentium-level machines.

Java in an Internet-Based Course

Our Operating Systems course (CS345) was very traditional: we used the "dinosaur" book (Silberschatz, 1997) and students had to extend the Nachos system (Christopher, 1997). The first step to export the course to the Internet was to transfer all overhead slides into PowerPoint presentations with animation and add the narration. We have augmented the presentations with three types of multimedia materials: still pictures, live video clips, and snapshots of handwritten materials from an electronic whiteboard. Live video clips are extremely large even with MPEG compression and have been used sparingly. We have found still pictures with an accompanying audio narration to be nearly as effective as video clips and require

BEST COPY AVAILABLE

considerably less space. We have found an electronic whiteboard to be an effective way to present certain types of materials. Although typed presentations are appropriate for exposition mode, they are less effective for a problem-solving mode. To capture a sequence of handwritten snapshots from a whiteboard with an accompanying audio narration is much more natural than to type the same solutions into PowerPoint. It also takes much less effort to develop the materials, particularly when equations or numerical solutions are involved.

The most interactive components we have added were developed in Java. In particular, we have developed a sequence of interactive modules covering major topics in the Operating Systems course. These modules have two modes of interaction: simulation where behavior is animated based on input data and exercise where the student must predict the behavior. Figure 5a shows the exercise mode for CPU scheduling.

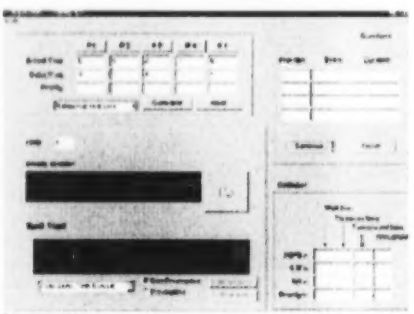
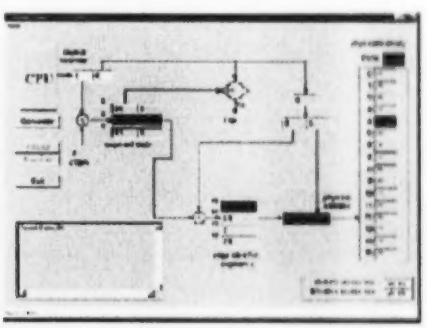


Figure 5a. CPU scheduling (exercise mode)

The upper left area is for input data that is either randomly generated or input by hand. The lower left section displays the scheduling process dynamically, showing both the ready queue and Gantt chart. In exercise mode the upper right area is used for student input of the next event; in simulation mode a scrollable trace of events is displayed. The lower right area shows statistical results after the simulation is completed; in exercise mode students enter requested values.

We have a variety of these modules throughout the course. Figures 5b, 5c, and 5d show screen snapshots for paged segmentation, page replacement, and disk scheduling.



BEST COPY AVAILABLE

Figure 5b. Paged segmentation (simulation mode)

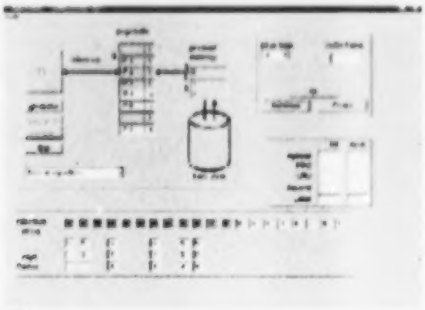


Figure 5c. Page replacement (exercise mode)

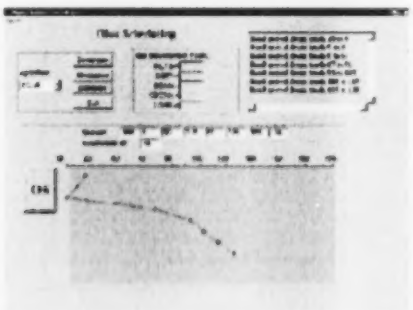


Figure 5d. Disk scheduling (simulation mode)

Java as a Programming Environment

For each of these projects we selected Java because of the language facilities, such as threads, and because the code is platform independent. We have found interfaces to be an acceptable substitute for multiple inheritance.

Java is a popular language that many regard as a cleaner version of C++. Compiling is extremely fast and the bytecode files are small. However, execution is interpreted and thus slower than native-code programs. But, Just-in-Time compilers are making this difference less noticeable. The newer programming environments, such as Symantec caf , Microsoft J++, and Borland JBuilder, substantially reduce project development time compared to the original Sun Java JDK. The language is very "clean" and does not result in errors commonly associated with C++, such as segmentation faults and core dumps.

Despite these benefits, we have also experienced difficulties. The most significant problems have been associated with the APIs. Although the core language is easy to master, particularly for those familiar with C++, the learning curve for the APIs is

very steep. Java is a recent language and still evolving. The transformation from version 1.0 to 1.1 was nontrivial. Event handling has changed significantly, and this required extensive recoding.

However, many of the new features in 1.1 solved problems that were plaguing us. For example, the editor in the Concurrency Simulator displays source code in string format, but the internal editor objects are Java classes created dynamically. In version 1.1 it is possible to save Java objects to files. Using Java 1.0 we could not save source files between sessions, but the transition to 1.1 solved this problem. Converting the event handling was painful, but the new mechanism is much more robust and acts more like callback functions expected in a point-and-click environment.

Another problem is platform independence. Although Java has made great strides in the development of a platform independent language that can be viewed on a variety of machines via the Internet, in reality the solution is far from perfect. There are a variety of small problems, such as color maps changing, and larger problems, particularly when the viewing environment is based on a different version of the language than the development environment. The evolution of Java will continue to cause some difficulties, but we are convinced we have made the correct language choice for these projects.

Issues Involving Development

Implementation of quality course materials using Java can easily demand more time than a faculty member has to give, even when state-of-the-art hardware and software are used. Faculty members often realize that they must turn to others (primarily students) for help. However, this "help" may backfire, with the training and coaching of the students taking more of the developer's time than developing the software directly. Factors that should be considered when working with students to develop Java software include (1) student ability and knowledge, (2) the learning curve, and (3) software and hardware resources available to the student.

It is important not to overwhelm the student initially; start off with a relatively easy assignment that will lead to success. We have found a simple calculator project provides a gentle, yet meaningful, introduction to Java. If you have multiple students using the same software environment, encourage peer learning as much as possible.

Students have different personalities and require different management styles. Some students perform best if they are allowed to use their creative talents while others require closer supervision and guidance. In particular, try not to suppress the creative student.

We have employed students on projects by either paying them cash (graduate assistantships or hourly wages) or "paying" them with credits (special topics courses or thesis credit). Paying with credits may appear to be cost effective but, in some cases, has not worked out well. Special topics courses are difficult to manage and difficult to judge when enough work has been completed, particularly when most students expect to get an A in such a course. Also, the time frame of a single quarter

or semester often does not provide sufficient time to both climb the learning curve and still get useful work out of the student. One approach that has worked fairly well is to tell prospective student workers who have little background in Java but who still wish the join the project that an up-front commitment of two quarters or semesters is required in order to earn credit.

Working for money gives more direct leverage over the student, but if the student's productivity begins to slip, it is often not a simple matter to replace him or her; again, the costs associated with the learning curve must be carefully weighted.

Regardless of which approach is adopted, careful supervision on the part of the faculty member leading the project is critical for success. It is a fallacy to believe that the faculty member can simply concentrate on the high-level design aspects of the project and leave it to students to produce reliable code. The faculty member must be willing to get his or her "hands dirty" and become directly involved with coding and quality assurance issues to produce a useful product.

Common Factors in the Design of Materials

Although we have presented a variety of Java projects, there are certain design features we have found to be important in each of these efforts. Perhaps the most important principle is to present a consistent interface. In the Watson labs the interface has a similar look and feel despite a wide range of topics covered. The Concurrency Simulator covers three different paradigms, but each has a similar interface. This is also true for each of the simulations developed for Operating Systems, despite the fact the topics are quite different.

Another principle is modularity—materials should be divided into small components that can be used by the students in a variety of ways. For example, in the Operating Systems course we include historical information, human-interest items, and frequently asked questions. These materials are "optional" and may be skipped by some students. The wide variety of labs in the CS100 course is neatly modularized and may be used in a many different ways. While it is important to allow the student the flexibility to explore different avenues of information, the student should be kept on track to ensure progress through the course materials.

A related factor is to allow for a variety of instructional approaches ranging from unguided exploration to self-test modes. Self-test mode is similar to simulation mode except each student can test his or her knowledge in a nonthreatening, nongraded environment. Students can check their approaches by switching back to simulation mode. We believe it is important to provide a self-test environment separate from quizzes and exams in the course.

Experience in the Classroom

The Watson labs are the oldest and the most mature of the projects discussed. The first field testing using labs developed in C was more than four years ago. Use of

these early labs helped us develop many of the educational principles that permeate all of these projects. The new Java-based labs have been introduced over the last year and a half.

Students seem to enjoy the labs. More importantly, these labs allow instructors to present "advanced" topics, such as digital circuit design, to naive audiences. Recent studies conducted by Rugg and O'Neal (Rugg, 1997) show, for example, that students given a data structures lecture alone scored on average 10%–15% lower than students who have received the lecture plus the data structures lab.

We are using the Concurrency Simulator for the second time and the results have been encouraging. Students using the context-sensitive editor have virtually no learning time to master the language syntax; the small learning time is associated with how to use the point-and-click environment. This allows students to concentrate on the logic of their algorithms and not on the syntax of the programming language. These results are similar to what is found in the graphics and imperative labs of the Watson project.

Using the topology as an integral part of the programming environment has been very helpful both in algorithm development, since many details can be handled by the topology itself, and in debugging the algorithm. The combination of the results window, status window, and changing colors on the topology makes it easy to spot and correct problems. We illustrate this by discussing the Dining Philosophers problem.

The Dining Philosophers program is subject to deadlock if initially every philosopher picks up the right fork before the left fork. One way to solve the deadlock is to have one philosopher pick up the forks in the opposite order from the other philosophers. In a text-based system this requires introduction of an if statement in the program to make sure one philosopher (usually the first or the last) picks up forks in the opposite order. This extra if statement starts making the algorithm difficult to understand. However, by integrating the topology into the program environment we have a very easy solution using the Concurrency Simulator: simply reverse the links on the topology and don't change the source code at all.

We have completed the initial development of two Internet-based courses: Operating Systems (CS345, described in this paper) and Data Structures. Both use Java applets to allow students to explore small problem domains and both are being tested for the first time in an Internet-based framework. It is too early to report on definitive results, but it is clear that Java adds student interaction to an otherwise static environment. Preliminary results on course evaluations indicate the a self-paced environment is suitable for some students, particularly nontraditional students or more mature students, but does not work so well with the typical 19- or 20-year-old student.

Advice for Potential Developers

Before embarking on a major effort to develop Java-based software it is critically

important that the administration realize not only the potential benefits for such development but also the costs. Estimates vary, but the cost to develop 1 hour of interactive computer-based materials ranges from 20 to 50 hours, depending on the level of simulations and the amount of intelligence built into the software. These are nontrivial costs that make initial development very expensive. These costs will decrease as you gain experience and the development tools become more sophisticated, but they will remain high for the near future. This is particularly true for computer science where the subject matter is continually evolving. We have been fortunate to have support from two NSF grants, but for those who do not have external support the school administration needs to be willing to contribute release time, student labor, hardware, and software if real progress is going to be made in developing Java-based educational materials. The youngest assistant professors are often the faculty most interested in developing such materials. It is critically important that the administration commit up-front to the acceptance of this work as valid applied research that would lead to a favorable tenure decision. Without such a commitment the best and the brightest will not be able to participate.

To develop any substantial amount of material, it is necessary to build up a cadre of student programmers. Several of the components described in this paper were parts of M.S. theses. Special topics courses can also be used with undergraduate students. Even if the projects are different it is important that these students work on sharing their knowledge about Java development. This is very important for overcoming the initial learning curve to master the library of APIs.

A significant challenge is to construct stable software that works reliably in an academic environment. Since one of the major purposes of these projects is to hide much of the complexity of real systems and languages—so that students can concentrate on problem solving rather than on learning the intricacies of real systems—student frustrations result when labs crash, are inconsistent, or display other behavioral problems.

As entering students have more and more exposure to graphical user environments, such as Windows 95, their levels of expectation as to reliability and ease of use continue to increase. It is truly a humbling experience to watch 30 kids discover several major flaws—all in less than one hour—in a system you have spent months developing and carefully testing. As Java matures and our code continues to be field tested in the lines of the classroom, stability is being achieved.

You must allow for extensive field testing of pedagogical effectiveness. We have found that some items that work well in the hands of an expert don't work so well for more naive students. You must be willing to modify (or even abandon) the software based on classroom testing. We have developed several Watson labs (e.g., Lisp programming) that we no longer use today because the programming environment was too difficult for students or the intended pedagogical goals were judged to have not been met.

Developing sophisticated Java-based software is a nontrivial task. You need to be willing to devote the necessary resources to the project. But once the software is fieldtested, modified, and tested again, you will find the effort worthwhile.

References

Andrews, G., & Olsson, R. (1993). *The SR programming language: Concurrency in practice*. Benjamin/Cummings.

Ben-Ari, M. (1982). *Principles of concurrent programming*. Prentice Hall.

BraŸnl, T. (1993). *Parallel programming: An introduction*. Prentice Hall International.

Christopher, W.A., Procter, S.J., & Anderson, T.E. *The Nachos instructional operating system*. http://cs-tr.cs.berkeley.edu/TR/UCB:CSD-93-739Anderson-nachos/.

Conway, M.J. (1992). *The SUIT version 2.3 reference manual*. University of Virginia.

Denning, P.J., Comer, D.E., Gries, D., Melder, M.C., Tucker, A., Turner, A.J., & Young, P.R. (1989, January). Computing as a discipline. *Communications of the ACM, 32(1), 9–23.*

Kurtz, B., Cai, H., Plock, C. & Chen, X. (1998a, February). A concurrency simulator designed for sophomore-level instruction. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education* (pp. 237–241). Atlanta.

Kurtz, B., Kim, C., & Alsabbagh, J. (1998b, February). Parallel computing in the undergraduate curriculum. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education* (pp. 212–216). Atlanta.

Kurtz, B., & O'Neal, M. (1994). An interdisciplinary, laboratory-based course for computer-based problem solving. *Proceedings of the National Educational Computer Conference*. Boston.

O'Neal, M., & Kurtz, B. (1995, March). *Watson: A modular software environment for introductory computer science education.* SIGCSE Technical Symposium on Computer Science Education, Nashville, TN.

Reps, T., & Teitelbaum, T. (1989). *The synthesizer generator: A system for constructing language-based editors*. Springer-Verlag.

Rugg, J. (1997). *A data structures lab for a breadth-first introduction to computer science*. Unpublished master's thesis, Louisiana Tech University.

Silberschatz, A., & Galvin, P.B. (1994). *Operating system concepts*. Addison-Wesley.

Slonneger, K., & Kurtz, B. (1995). *Formal syntax and semantics of programming languages*. Addison Wesley.